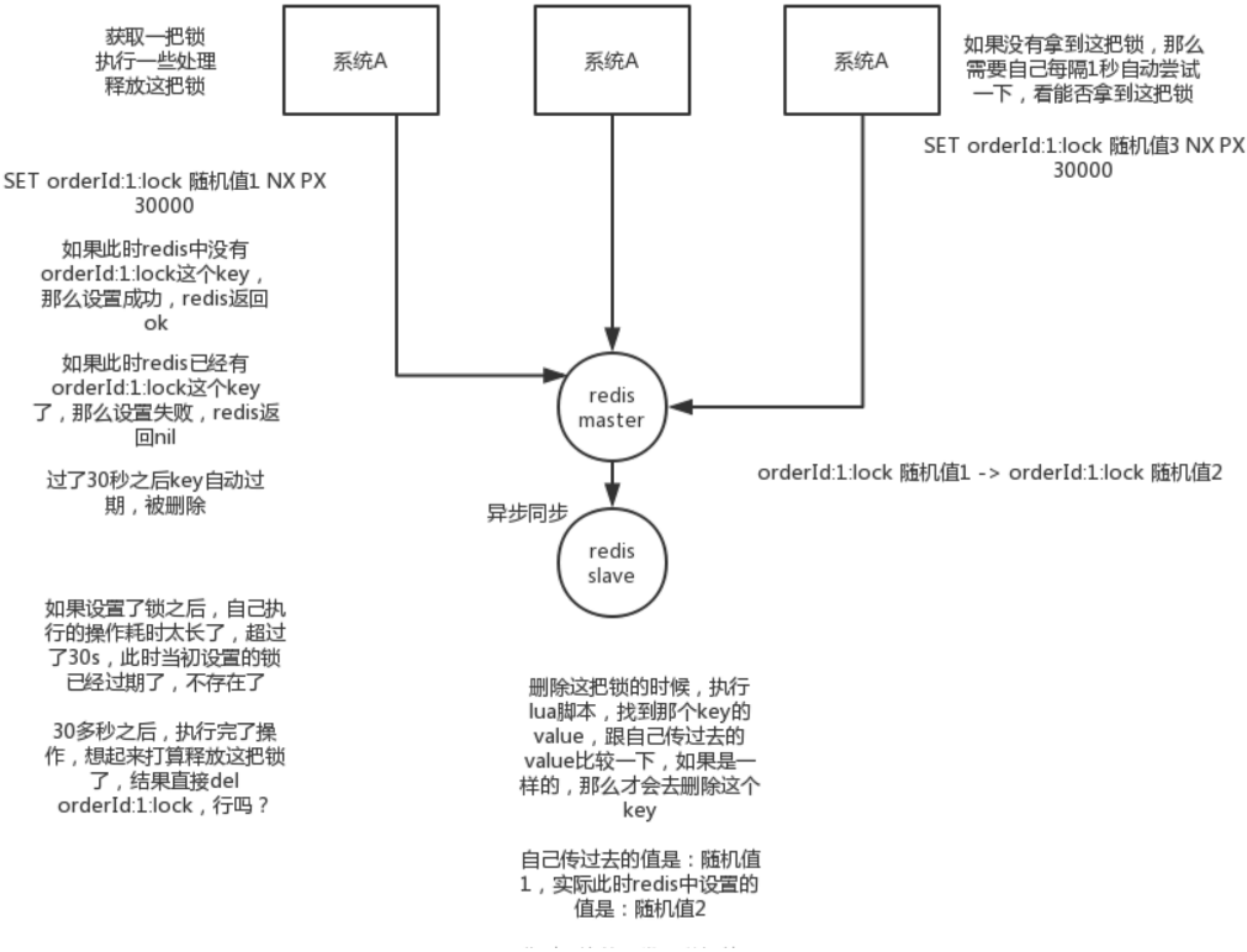
## Redis分布式锁




- 最普通的实现方式

SET my:lock 随机值 NX PX 30000，这个命令就ok，这个的NX的意思就是只有key不存在的时候才会设置成功，PX 30000的意思是30秒后锁自动释放。别人创建的时候如果发现已经有了就不能加锁了。

释放锁就是删除key，但是一般可以用lua脚本删除，判断value一样才删除：

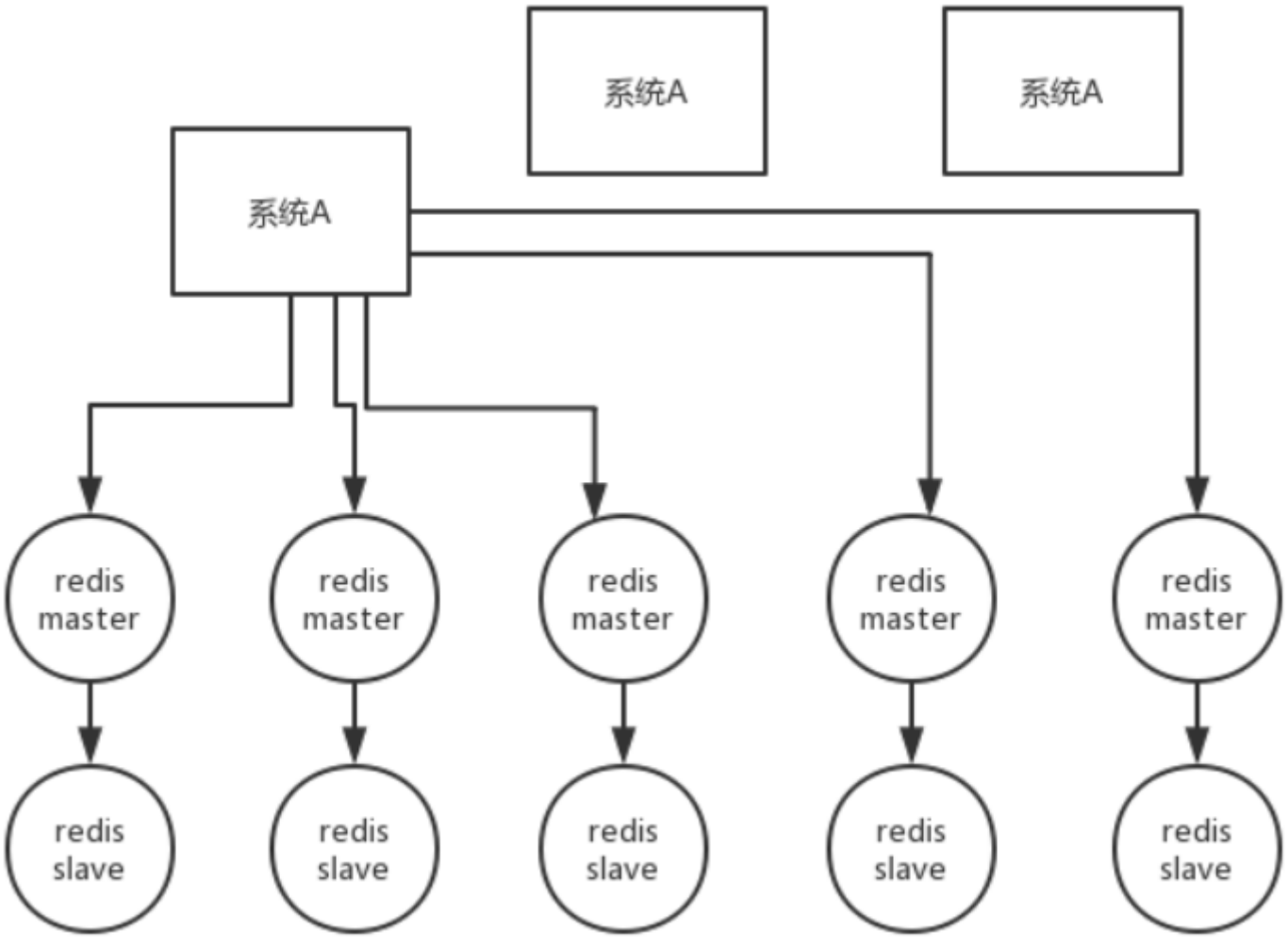
关于redis如何执行lua脚本，自行百度

```
if redis.call(get,KEYS[1]) ==
ARGV[1] then
    return
redis.call(del,KEYS[1])
else
return 0
end
```

为啥要用随机值呢？因为如果某个客户端获取到了锁，但是阻塞了很长时间才执行完，此时可能已经自动释放锁了，此时可能别的客户端已经获取到了这个锁，要是你这个时候直接删除key的话会有问题，所以得用随机值加上面的lua脚本来释放锁。

存在的问题：如果是普通的redis单实例，那就是单点故障。或者是redis普通主从，那redis主从异步复制，如果主节点挂了，key还没同步到从节点，此时从节点切换为主节点，别人就会拿到锁

- RedLock算法




这个场景是假设有一个redis cluster，有5个redis master实例。然后执行如下步骤获取一把锁：

1）获取当前时间戳，单位是毫秒

2）跟上面类似，轮流尝试在每个master节点上创建锁，过期时间较短，一般就几十毫秒

3）尝试在大多数节点上建立一个锁，比如5个节点就要求是3个节点（n / 2 +1）

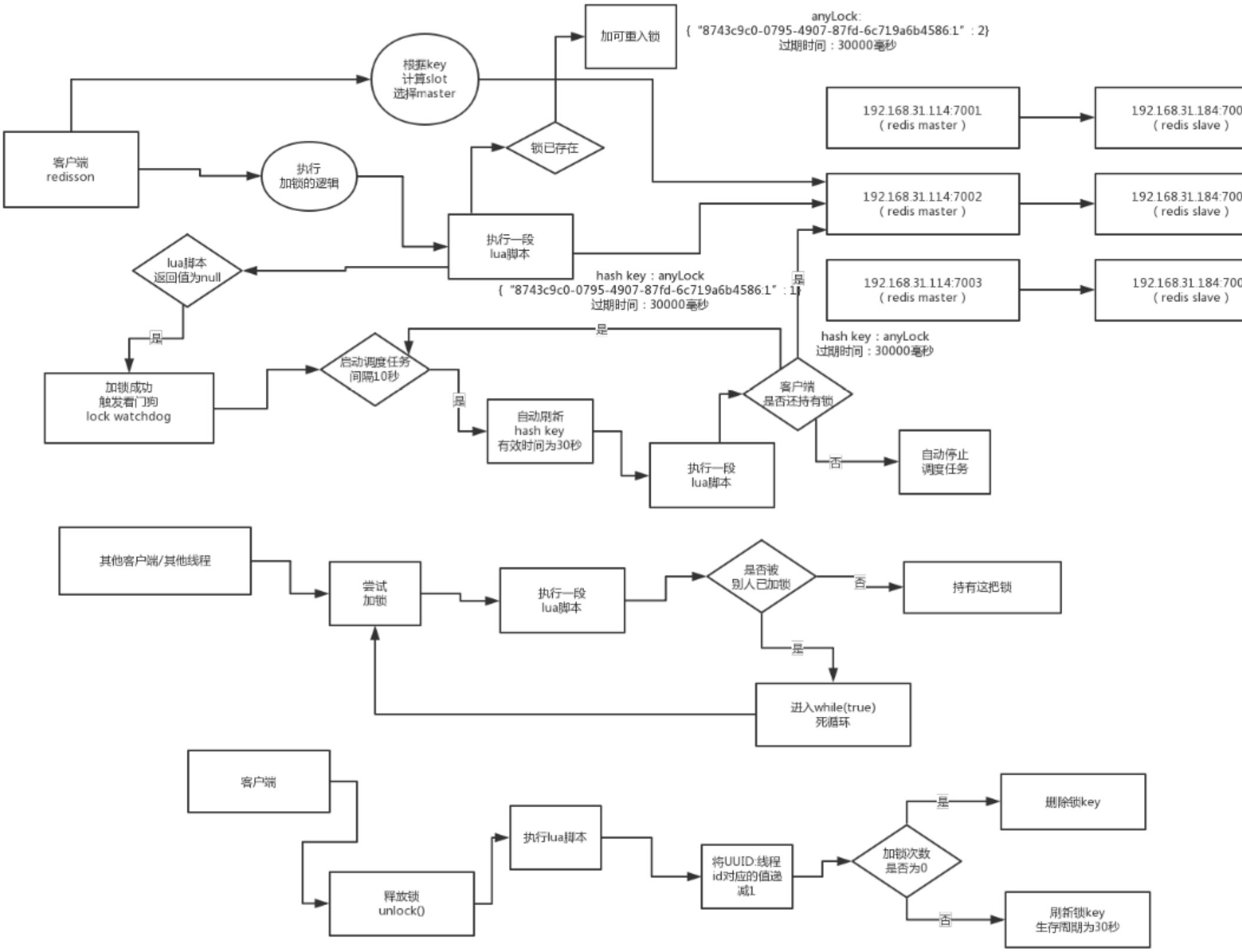
4）客户端计算建立好锁的时间，如果建立锁的时间小于超时时间，就算建立成功了

5）要是锁建立失败了，那么就依次删除这个锁

6）只要别人建立了一把分布式锁，你就得不断轮询去尝试获取锁

存在的问题：基于redis主从架构+哨兵，保证高可用，master宕机，slave接替，但是有隐患，master宕机的一瞬间，还没异步复制锁到slave，导致重复加锁的问题，高可用是高可用了，但是锁的实现有漏洞，可能导致系统异常，客户端刚在master实例加了一个锁，但是master->slave的复制数据（锁复制过去）是异步的，导致master突然宕机，此时锁还没复制到slave，然后master->slave主备切换（哨兵），客户端B此时也对同一个key上锁，此时就会成功的在切换为master的slave实例上加锁，客户端A和客户端B同时对一个key完成了上锁

## Redisson




- Redisson lock底层原理

```
if (redis.call('exists', KEYS[1]) ==0) then  +
        redis.call('hset',KEYS[1], ARGV[2], 1);  +
        redis.call('pexpire', KEYS[1], ARGV[1]);  +
                return nil; +
                end;  +
                if(redis.call('hexists', KEYS[1], ARGV[2]) == 1) then  +
                    redis.call('hincrby', KEYS[1],ARGV[2], 1);  +
                    redis.call('pexpire', KEYS[1], ARGV[1]);  +
                    return nil; +
                end;  +
                return redis.call('pttl', KEYS[1]);
```

其中，KEYS是一个数组，[anyLock]，这个其实对应的就是上面分析的那个脚本里的KEYS[1]，params，是一个参数，[30000，8743c9c0-0795-4907-87fd-6c719a6b4586:1]，代表的其实就是ARGV[1]和ARGV[2]

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = 8743c9c0-0795-4907-87fd-6c719a6b4586:1

8743c9c0-0795-4907-87fd-6c719a6b4586，代表的是一个UUID，其实就是这个客户端上的一个ConnectionManager的这么一个id，1是什么呢？1大家还记得吧，其实就是threadId，大体上可以认为是一个客户端的一个线程对应的唯一的标识

ARGV[2]，代表的就是一个客户端上的一个线程，对这个key进行了加锁

hset anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

anyLock就对应一个map数据结构（hash）

```
{
        “8743c9c0-0795-4907-87fd-6c719a6b4586:1”:1
}
```

pexpire anyLock 30000：anyLock这个key只能存活30000毫秒，30秒，默认情况下，你使用redisson加锁，其实不是无限制的不停的可以拥有这把锁的，人家默认情况下给你设置的一个锁的有效期就是30秒，watchdog看门狗的代码的实现，人家一定是在不断的监控这个锁，如果到30秒，就自动释放掉这把锁，或者是自动延期这个key的有效时长

redisson实现了一个看门狗，推测，根据文档里面的东西来推测，可能这个看门狗到了30秒，就自动把这个key给延期了，在延期的时候，他可能是怎么做的呢？如果看门狗在30秒内再次执行了这个加锁的方法，此时很可能会走到下面的那段脚本（可重入锁）

```
if(redis.call('hexists', KEYS[1], ARGV[2]) == 1) then  +
    redis.call('hincrby',KEYS[1], ARGV[2], 1);  +
    redis.call('pexpire', KEYS[1], ARGV[1]);  +
    return nil; +
end;  +
```

hexists anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1：判断一下，针对anyLock这个map，里面是否有一个8743c9c0-0795-4907-87fd-6c719a6b4586:1 key，如果有，就说明这个客户端的那个线程，对这个key之前是加过锁的，而且此时还持有锁，因为key还没过期

hincrby anyLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1，针对anyLock这个map中的8743c9c0-0795-4907-87fd-6c719a6b4586:1这个key的值，累加1

anyLock就对应一个map数据结构（hash）

```
{
    “8743c9c0-0795-4907-87fd-6c719a6b4586:1”:2
}
```

这个值对应的数字就代表了，看门狗延长过多少次这个key的有效时长

pexpire anyLock 30000，再次将这个anyLock这个key的有效时长设置为了30000毫秒，30秒

如果客户端一直长期持有了一把锁，还没释放，但是redis里的key对应的有效期是30秒，看门狗会自动去延长那个key的有效期，更新一下延长的次数，只有当你的客户端手动释放unlock锁的时候，才会删除那个key

- 其他线程加锁阻塞过程

如果说客户端A已经上锁了，还持有着这把锁，此时客户端B尝试加锁，此时就会直接执行pttl anyLock指令，返回这个key剩余的一个存活时间，ttlRemaining不是null，说明加锁没成功，就不会启动后台的定时调度任务每隔10秒去刷新锁key的生存周期

```
long threadId =Thread.currentThread().getId();
 Long ttl = tryAcquire(leaseTime, unit, threadId);
 // lock acquired
 if (ttl == null) {
        return;
 }
```

tryAcquire()方法尝试加锁，获取到一个ttl，如果是一个线程第一次加锁，ttl一定是null；如果是一个线程多次加锁，可重入锁的概念，此时ttl也一定是null，lua脚本里返回的就是nil；但是如果加锁没成功，锁被其他机器占用了，你执行lua脚本直接获取到的是这个key对应的剩余生存时间

如果ttl是null，证明加锁成功，就会直接返回

如果ttl不是null，证明加锁不成功，此时就会走阻塞逻辑

```
while (true) {
        ttl = tryAcquire(leaseTime,unit, threadId);
        // lock acquired
        if (ttl == null) {
            break;
        }

        // waiting for message
        if (ttl >= 0) {
            getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
        } else {
            getEntry(threadId).getLatch().acquire();
        }
 }
```

如果加锁不成功，直接会进入while(true)就是一个死循环内，在死循环内，再次执行这个ttl =tryAcquire(leaseTime, unit, threadId);尝试去获取这个分布式的锁，如果获取到了锁，证明ttl是null，此时就会退出死循环，如果ttl大于等于0，说明其他的客户端还是占据着这把锁

```
if (ttl >= 0) {
        getEntry(threadId).getLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
 } else {
        getEntry(threadId).getLatch().acquire();
```

```
        }
```

如果获取锁不成功，此时就会等待一段时间，再次投入到while(true)死循环的逻辑内，尝试去获取锁，以此循环往复，看到这样的一个分布式锁的阻塞逻辑，如果一个客户端的其他线程，或者是其他客户端的线程，尝试获取一个已经被加锁的key的锁，就会在while(true)死循环里被无限制的阻塞住，无限制的等待，尝试获取这把锁

- 释放锁原理

```
if (redis.call('exists', KEYS[1]) ==0) then  +
            redis.call('publish',KEYS[2], ARGV[1]);  +
            return 1;  +
        end; +
        if (redis.call('hexists',KEYS[1], ARGV[3]) == 0) then  +
            return nil; +
        end;  +
        local counter =redis.call('hincrby', KEYS[1], ARGV[3], -1);  +
        if (counter > 0) then +
            redis.call('pexpire',KEYS[1], ARGV[2]);  +
            return 0;  +
        else  +
            redis.call('del',KEYS[1]);  +
            redis.call('publish',KEYS[2], ARGV[1]);  +
            return 1; +
    end;  +
            return nil;
```

如果anyLock这个key不存在，publish redisson_lock__channel_anyLock 0，redis是支持发布/订阅模型的，就是说可以对里面的某个channel 。key进行订阅，订阅他的消息，如果别人发布了消息在这个channel key里，别人就可以监听到。hexists anyLock 26cebeaa-e3b0-4097-8192-d62d0d0214b8:1，也就是说判断一下当前这个锁key对应的hash数据结构中，是否存在当前线程加的这个锁？释放锁的这个线程，是否之前加过锁？如果当前这个线程确实对这个锁进行了加锁，此时就会做一个处理，hincrby anyLock 26cebeaa-e3b0-4097-8192-d62d0d0214b8:1，给递减1，递减完了以后的值，是一个counter，计数器，可重入锁，一个线程可以多次加锁，线程id对应的counter，加锁的次数，先给加锁的次数减1,如果此时线程只加过一次锁，此时26cebeaa-e3b0-4097-8192-d62d0d0214b8:1对应的值本来是1，此时递减1，此时就是对应的counter = 0，就会删除这个锁key，del anyLock，删除这个锁key，释放掉这个锁，返回的值是1。如果此时线程加过多次锁，此时26cebeaa-e3b0-4097-8192-d62d0d0214b8:1对应的值是2，说明这个线程加过两次锁，可重复锁，递减1之后，还是1，大于0，pexpire anyLock 30000，刷新一下这个锁key的生存周期是30秒，返回的值是0。

- 锁超时释放

tryLockInnerAsync()，如果你自己指定了一个leaseTime，就会直接执行lua脚本去加锁，加完锁的结果就直接返回了，并不会对那个future加一个监听器以及执行定时调度任务去刷新key的生存周期，因为你已经指定了leaseTime以后，就意味着你需要的是这个key最多存在10秒钟，必须被删除，我们的那个锁到了10秒钟，就会自动被redis给删除。

- 总结Resisson加锁原理

（1）加锁：在redis里设置hash数据结构，生存周期是30000毫秒

（2）维持加锁：代码里一直加锁，redis里的key会一直保持存活，后台每隔10秒的定时任务（watchdog）不断的检查，只要客户端还在加锁，就刷新key的生存周期为30000毫秒

（3）可重入锁：同一个线程可以多次加锁，就是在hash数据结构中将加锁次数累加1

（4）锁互斥：不同客户端，或者不同线程，尝试加锁陷入死循环等待

（5）手动释放锁：可重入锁自动递减加锁次数，全部释放锁之后删除锁key

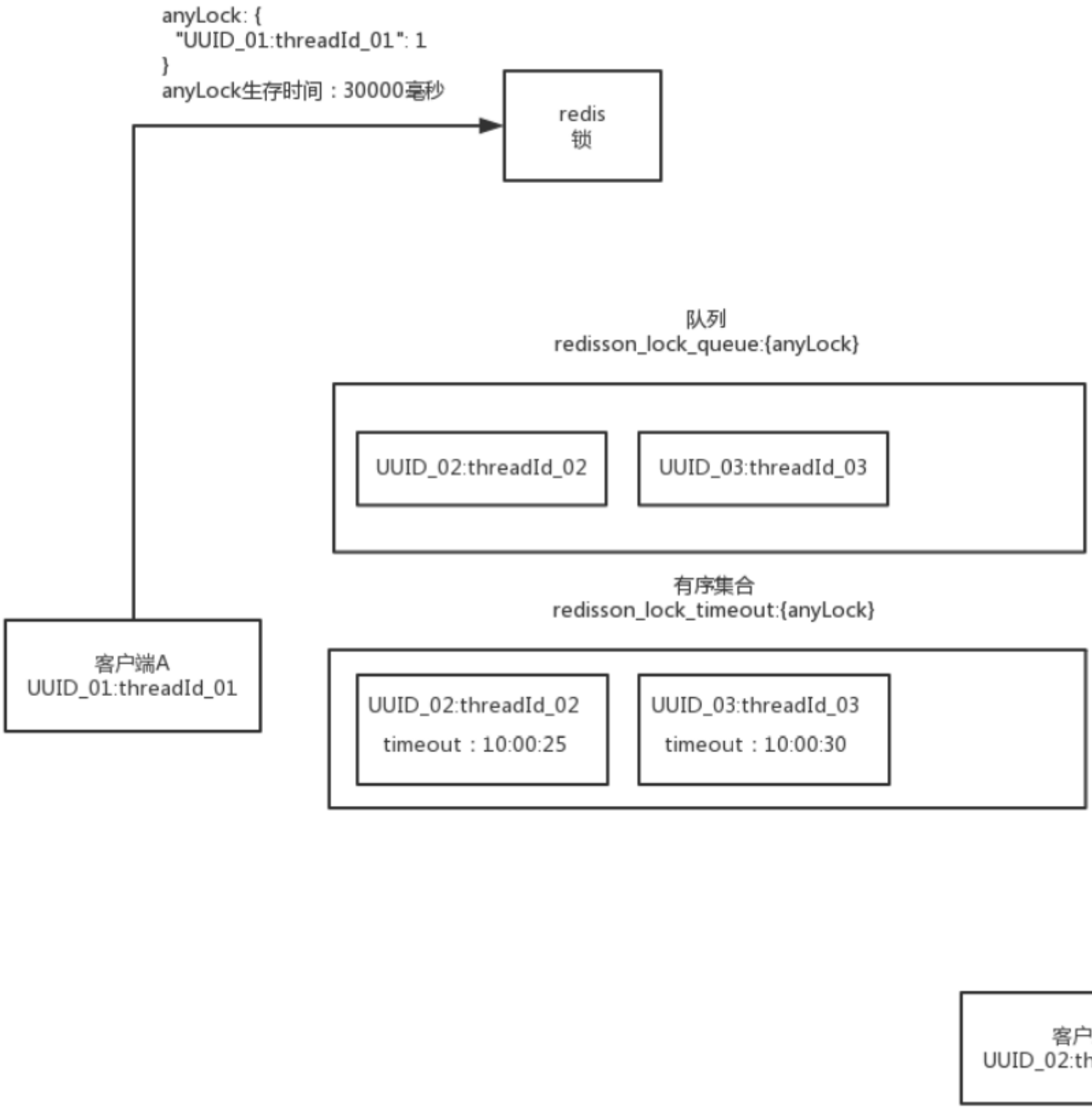
（6）宕机自动释放锁：如果持有锁的客户端宕机了，那么此时后台的watchdog定时调度任务也没了，不会刷新锁key的生存周期，此时redis里的锁key会自动释放

（7）尝试加锁超时：在指定时间内没有成功加锁就自动退出死循环，标识本次尝试加锁失败

（8）超时锁自动释放：获取锁之后，在一定时间内没有手动释放锁，则redis里的key自动过期，自动释放锁

存在的问题：redis加锁，本质，还是在redis集群中挑选一个master实例来加锁，master -> slave，实现了高可用的机制，如果master宕机，slave会自动切换为master，假设客户端刚刚在master写入一个锁，此时发生了master的宕机，但是master还没来得及将那个锁key异步同步到slave，slave就切换成了新的master。此时别的客户端在新的master上也尝试获取同一个锁，会成功获取锁，此时两个客户端，都会获取同一把分布式锁，可能有的时候就会导致一些数据的问题

- 多个客户端加锁公平锁原理




（1）客户端B和C如果获取不到锁，就会进入队列和有序集合，有序集合根据timeout值设置分数

（2）可重入锁是通过刷新timeout的值更新实现的

（3）释放锁：客户端A他释放锁的时候，也会走while true的脚本逻辑，看一下有序集合中的元素的timeout时间如果小于了当前时间，就认为他的那个排队就过期了，就删除他，让他后面重新尝试获取锁的时候重排序。while true的逻辑，比如说客户端B或者客户端C，他们用的是tryAcquire()方法，他们其实设置了一个获取锁超时的时间，比如说他们在队列里排队，但是尝试获取锁超过了20秒，人家就不再尝试获取锁了，此时他们还是在队列和有序集合里占了一个坑位，while true的逻辑就可以保证说剔除掉这种不再尝试获取锁的客户端，有序集合里的timeout分数就不会刷新了，随着时间的推移，肯定就会剔除掉他。如果客户端宕机了，也会导致他就不会重新尝试来获取锁，也就不会刷新有序集合中的timeout分数，不会延长timeout分数，while true的逻辑也可以剔除掉这种宕机的客户端在队列里的占用。因为网络延迟等各种因素在里面，可能会在等待锁时间过长的时候，触发各个客户端的排队的顺序的重排序，有的客户端如果在队列里等待时间过长了，那么其实是可以触发一次队列的重排序的。

（4）队列中的客户端是如何加锁的

如果客户端A释放了锁，删除了锁key之后，假设客户端C在队列里面是排在后面的，假设锁被释放掉了之后，是客户端C先拉尝试加锁，先判断下列逻辑

exists anyLock = 0，当前锁不存在；exists redisson_lock_queue:{anyLock} = 0，要不然就是队列不存在，但是现在队列是存在的；lindex redisson_lock_queue:{anyLock} 0 = UUID_02:threadId_02，队列存在，排在队头的是客户端C也可以，但是上面的逻辑不存在，刷新客户端C在有序集合中timeout分数，此时客户端B来加锁，anyLock锁key不存在的；队列是存在的；队列的队头就是客户端C，所以此时加锁的条件成立了，进入加锁的逻辑，将队列中的第一个元素弹出来，完成加锁逻辑。

- MultiLock

将多个锁合并为一个大锁，对一个大锁进行统一的申请加锁以及释放锁，一次性锁定多个资源，再去处理一些事情，然后事后一次性释放所有的资源对应的锁，比如说锁掉一个库存，锁掉一个订单，锁掉一个积分，一次性锁掉多个资源，多个资源都不让别人随意修改，然后你再一次性更新多个资源，释放多个锁。

获取锁是通过 while (true) {

```
if (tryLock(waitTime, leaseTime, unit)) {
    return;
}
}
```

不停的尝试去获取到所有的锁，只有获取到所有的锁的时候，while true死循环才会退出

lockAcquired = lock.tryLock(awaitTime,newLeaseTime, TimeUnit.MILLISECONDS);

lock是底层的RedissonLock，他没有使用lock.lock()，用的是tryLock()，指定了获取锁等待超时的时间，4500毫秒，必须获取到这个锁，如果获取不到这个锁，就退出，标记为获取锁失败；如果获取到锁之后，这个锁在多长时间内会自动释放，newLeaseTime是-1，因为你的newLeaseTime是-1，所以说如果获取到了锁，会启动一个lock watchdog不断的刷新你的锁key的生存时间为30000毫秒；默认的行为之下，你包裹了几把锁，就会锁数量 *1500毫秒，获取所有的锁必须在多长时间之内就要结束，如果超时就会重新再次死循环尝试获取锁。使用的是各个锁的tryLock()方法，指定了说在获取每个单独的锁的时候，会有一个获取超时退出的时间。

- RedLock

RedissonRedLock锁的实现，非常的简单，他是RedissonMultiLock的一个子类，RedLock算法的实现，是依赖于MultiLock的一个机制来实现的

```
@Override
protected int failedLocksLimit() {
return locks.size() - minLocksAmount(locks);
        }
protected int minLocksAmount(final List<RLock> locks) {
return locks.size()/2 + 1;
        }
@Override
protected long calcLockWaitTime(long remainTime) {
    return Math.max(remainTime / locks.size(), 1);
      }
```

主要就是通过方法的重载，改变了MultiLock中的几个特殊的行为，failedLocksLimit 最小加锁失败的数量是锁的个数locks.size()-最小加锁数量minLocksAmount，minLocksAmount是n / 2 + 1，calcLockWaitTime这个东西算出来的时间，是说在对每个lock进行加锁的时候，有一个尝试获取锁超时的时间，原来默认的就是remainTime，4500毫秒，4500毫秒 / 3 = 1500毫秒，每个小lock获取锁超时的时间改成了1500毫秒，对每个小lock尝试加锁的时候，能够容忍的最大超时时间，就是1500毫秒，1500毫秒之内必须加成功这个小锁，否则就是加锁失败。RedLock，是一个锁，只不过是在各个不同的master实例上进行加锁，但是现在说RedLock合并了多个小lock。也就是说，如果你有3个redis master实例，你就用lock1、lock2、lock3三个锁key，人家本来就是分布在3个不同的redis master实例上的加这个RedLock，相当于是3个redis master实例上的key，有2个加成功了，就算这个RedLock加锁成功了。

- 读写锁

（1）RedissonReadLock

加读锁时，key anyLock: { “mode”:“read”，“UUID_01:threadId_01”:1} 对应一个hash 同时会设置{anyLock}:UUID_01:threadId_01:rwlock_timeout:1　　1，同时会开启一个watchdog看门狗，会每隔10秒钟去执行一段lua脚本，判断一下当前这个线程是否还持有着这个锁，如果还持有锁，更新一下锁key的生存时间为30000毫秒，保持redis的锁key和java代码中持有的锁是保持同步的。

读锁的watchdog的逻辑：首先获取一下当前这个线程是否对这个锁加了一个读锁，这里返回的应该是1，此时可以判定是当前这个线程加的读锁，刷新一下anyLock锁key的生存时间为30000毫秒，加读锁的时候，其实是每个线程都可以加多次这个读锁，读锁也是可重入的，每次同一个线程加多次读锁的时候，他的加锁次数就会加1。watchdog在刷新生存周期的时候，一方面是说，如果这个线程对这个key还加着锁的话，那么此时就会刷新锁key的生存周期，anyLock，30000毫秒；同时还会遍历加锁次数，对那个锁key的每次加锁对应的一个timeout key也把生存周期刷新为30000毫秒。

（2）RedissonWriteLock

KEYS[1] = anyLock

ARGV[1] = 30000

ARGV[2] = UUID_01:threadId_01:write

hget anyLock mode，此时肯定是没有的，因为根本没这个锁

hset anyLock mode write

hset anyLock UUID_01:threadId_01:write 1

pexpire anyLock 30000

```
anyLock: {
  “mode”:“write”,
  “UUID_01:threadId_01:write”:1}
```

加写锁就已经成功了

（3）读锁与读锁非互斥

```
anyLock: {
                “mode”:“read”,
                “UUID_01:threadId_01”:1,
                 “UUID_02:threadId_02”:1
}
```

{anyLock}:UUID_01:threadId_01:rwlock_timeout:1 1

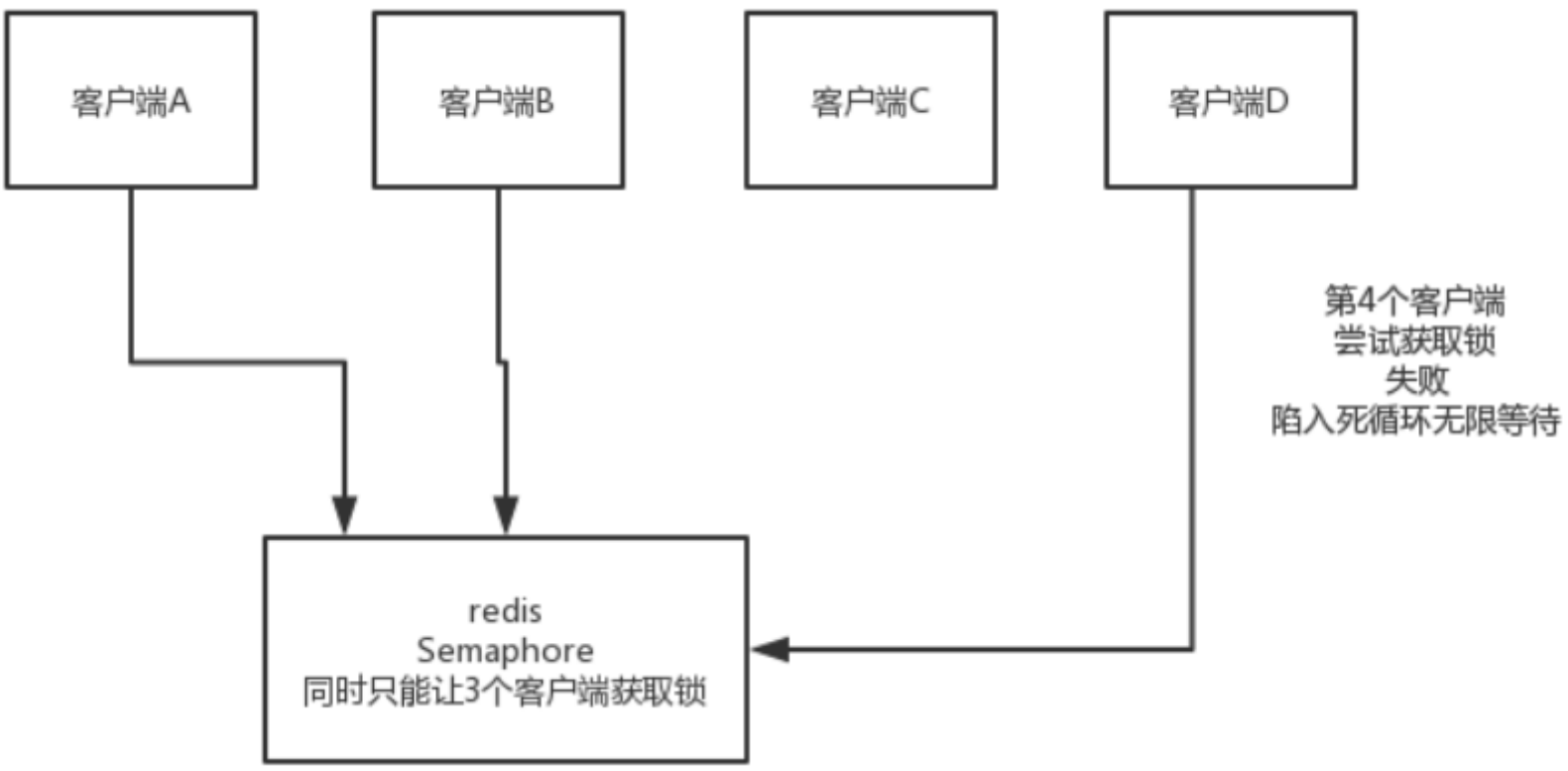
{anyLock}:UUID_02:threadId_02:rwlock_timeout:1 1

多个客户端，同时加读锁，读锁与读锁是不互斥的，只会让你不断的在hash里加入哪个客户端也加了一个读锁

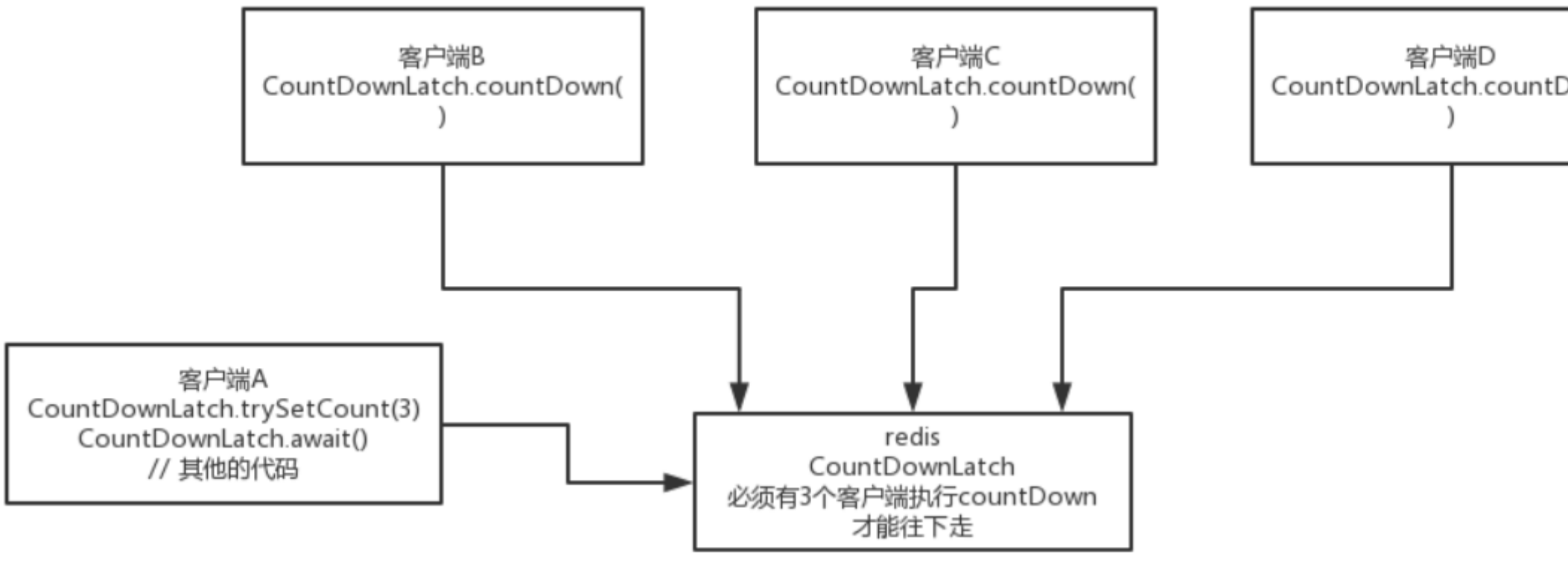
每个客户端都会维持一个watchdog，不断的刷新anyLock的生存时间，同时也会刷新那个客户端自己对应的timeout key的生存时间

（4）不同客户端，读锁与读锁不互斥，读锁与写锁互斥，写锁与读锁不互斥，写锁与写锁不互斥

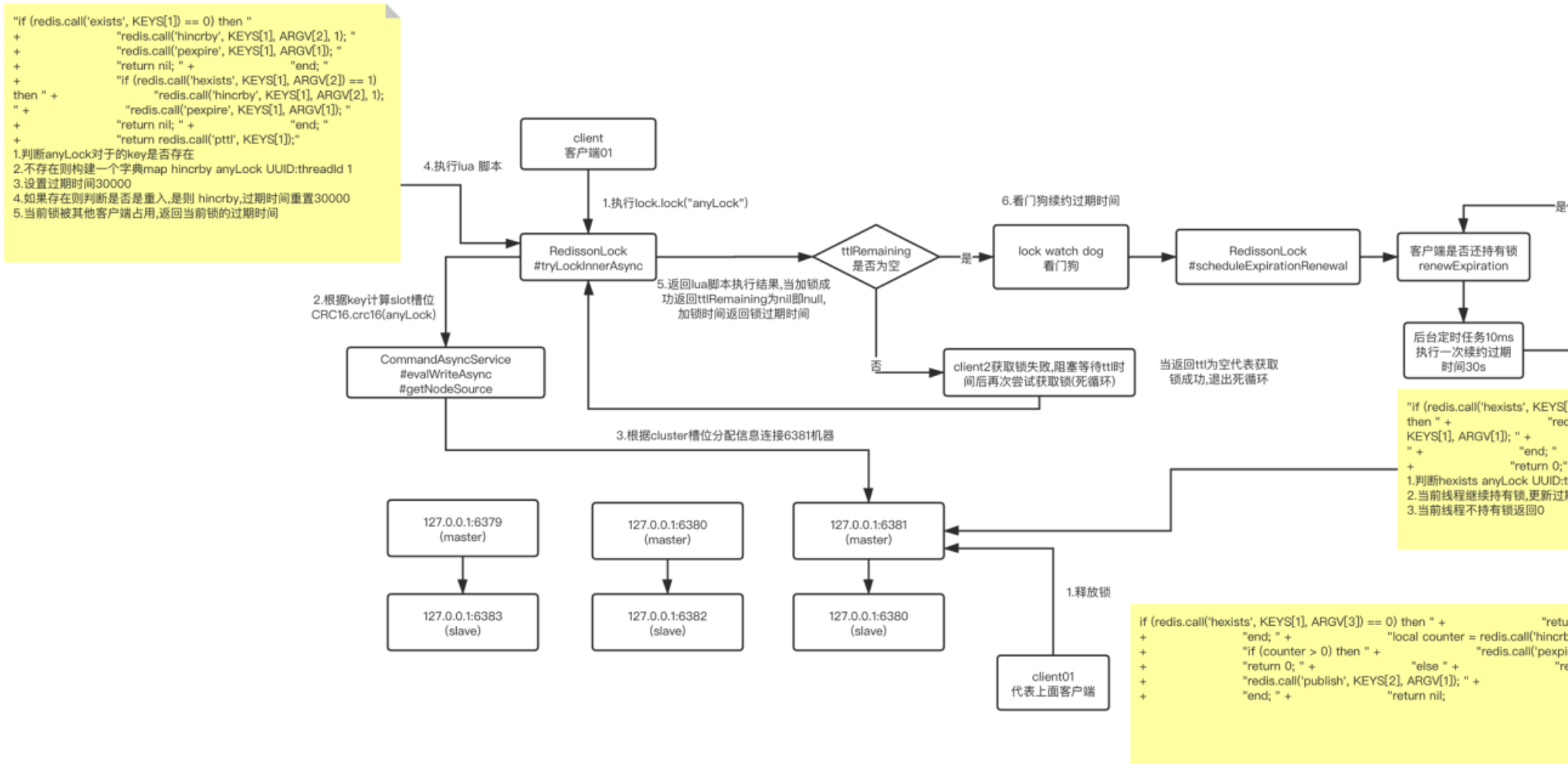
- Semaphore



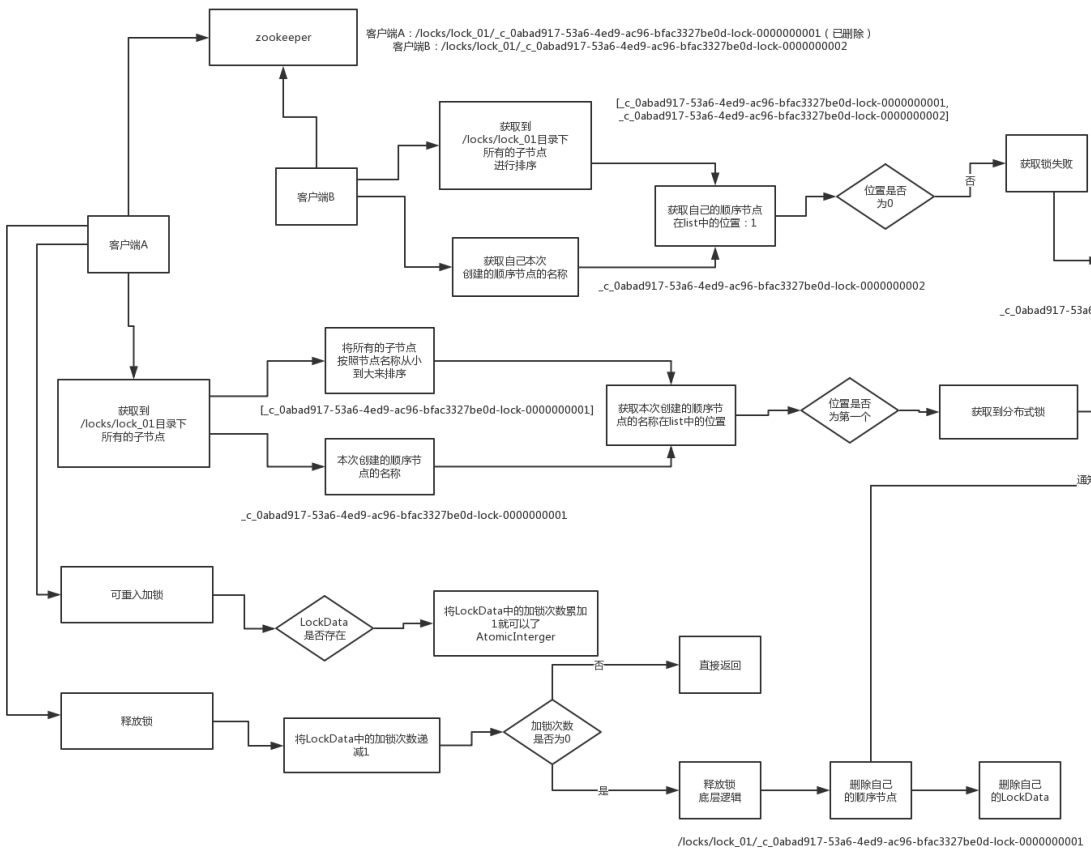

- CountDownLatch



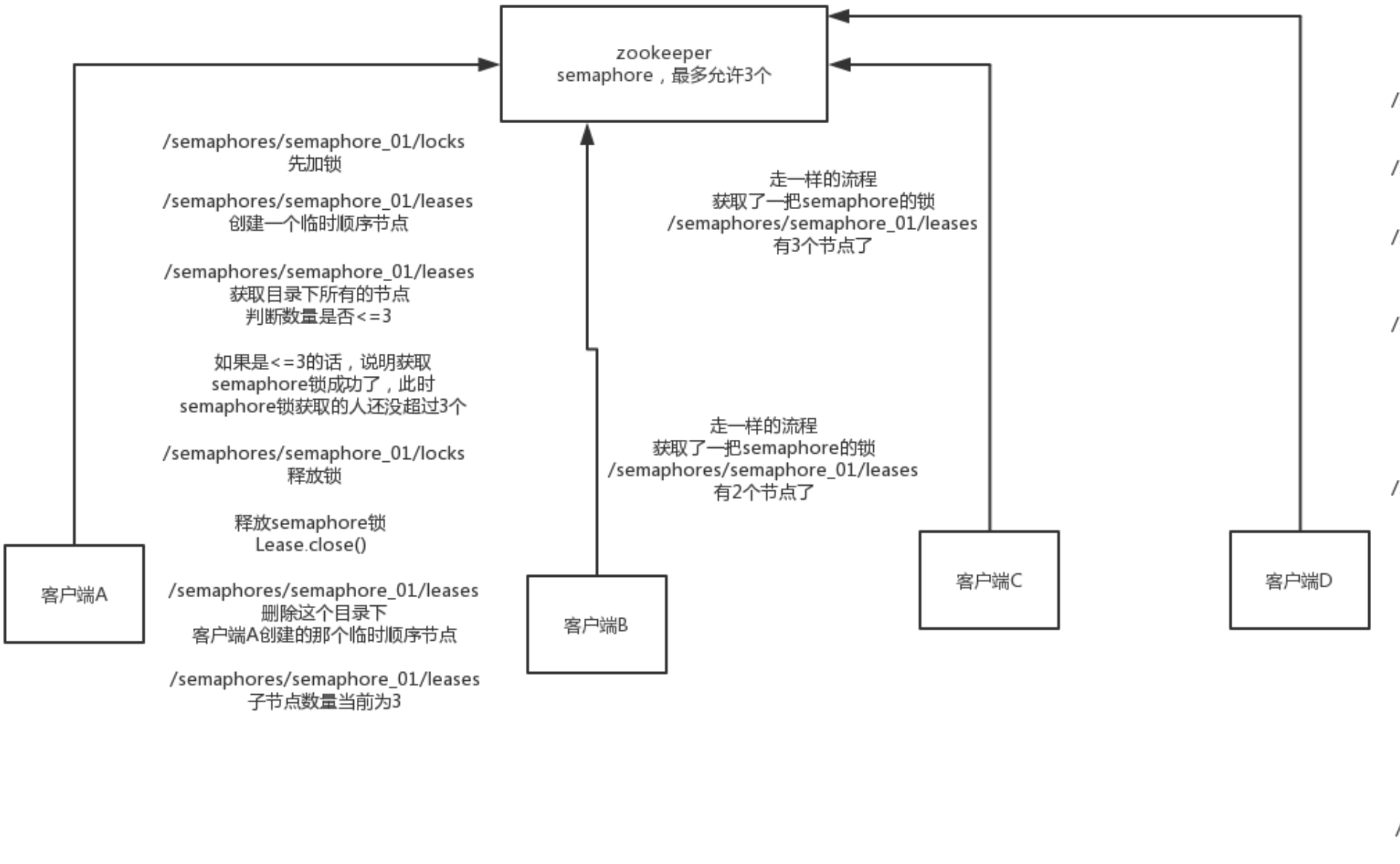

- Redisson总结




# Zookeeper分布式锁




- Semaphore源码




- 非可重入锁源码

  跟可重入锁其实是一样的，但是只不过没有提供可重入的支持，同一个时间只能有一个客户端获取到锁，其他人都是要排队的，同一个客户端是不可重入加锁的，基semaphore来实现的，只不过将semaphore允许获取锁的客户端的数量设置为了1，同一时间只能有一个客户端获取到锁

- 可重入读写锁

  （1）读锁+读锁（不互斥）

  每次去加一个读锁的时候，都是在/locks/lock_01目录下创建一个顺序节点，然后，获取一个顺序节点在/locks/locks_01目录下的位置，索引，判断这个位置只要< Integer.MAX_VALUE这个值就可以了，N多个客户端同时加读锁，肯定是不会互斥的。

  （2）读锁+写锁（互斥）

  /locks/lock_01目录下，此时已经有了一个顺序节点，有了N个读锁的顺序节点，/locks/lock_01/_c_0548a389-3307-4134-9551-088d305b86c7-__READ__0000000003，就代表了说一个客户端加了读锁，/locks/lock_01/_c_73b60882-9361-4fb7-8420-a8d4911d2c99-__WRIT__0000000005，加写锁的时候，直接在/locks/locks_01目录下创建一个__WRITE__的写锁的顺序节点，此时/locks/lock_01目录下，之前已经有人加过一个读锁了，此时又往里面写了一个写锁的顺序节点[_c_13bf63d6-43f3-4c2f-ba98-07a641d351f2-__READ__0000000004,_c_73b60882-9361-4fb7-8420-a8d4911d2c99-__WRIT__0000000005]，写锁节点在children里是排在第二位的，index是1，写锁的maxLeases是1，所以说如果你要加一个写锁成功的话，你必须是在/locks/lock_01目录里，是处于第一个位置的，index = 0，才能小于maxLeases，写锁才能够加成功，但是此时children中，第一个的是别人加的读锁，所以此时你的写锁一定是失败的，他会给他的前一个节点加一个监听器，如果前面一个节点释放了锁，他就会被唤醒，再次尝试判断，他是不是处于处于当前这个children列表中的第一个，如果是第一个的话，才能是加写锁成功。

  （3）写锁+读锁

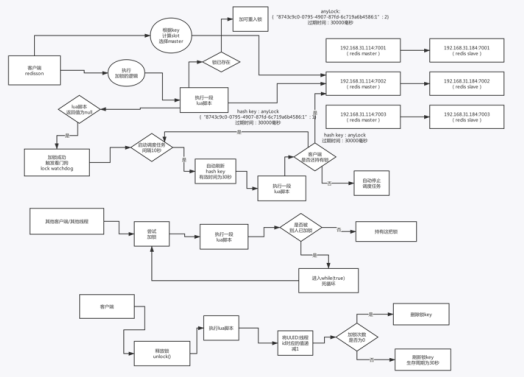
  同一个客户端，先加写锁再加读锁是可以成功的，不同的客户端，先加写锁再加读锁是互斥的。因为同一个客户端读锁只需要判断/locks/locks_01目录下的位置<Integer.MAX_VALUE这个值就可以了,但是不同客户端会先判断，如果先加写锁再加读锁，会卡住，进入wait状态，只有等写锁释放之后唤醒后面的读锁。

  （4）写锁+写锁

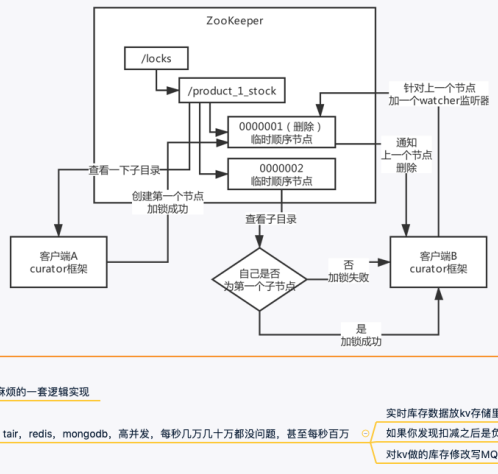
  如果有一个人先加了写锁，然后后面又有一个人来加了这个写锁，此时会发现第二个写锁的node是第二位，不是第一位，所以会导致写锁也会等待，加锁失败，只有第一个写锁先成功了，第二个写锁才能成功

- 分布式锁总结

- 分布式锁
  - Redis分布式锁
    - Redisson框架
      - 基于redis实现了一系列开箱即用的高级功能，比如分布式锁
      - 原理
        - 加锁
          - 客户端 → 向集群某一master节点，发送lua脚本，执行加锁逻辑lock(key,time) → lua脚本
            - 加锁 → 可重入，下次再加锁时，lua脚本判断客户端id是不是自己的，是的话支持重入，加锁次数+1
            - 开启watchdog看门狗 → 这是个后台线程，每 time/3 s执行一次，刷新锁的过期时间 → 客户端宕机时，看门狗线程被销毁，key按ttl过期消失
          - 基于redis m-s集群模式，可能会出现问题
            - 若m-s模式中，m刚加完锁未来的及同步到
            - s重新成为m时，刚刚的锁丢了
            - 其他客户端执行加锁逻辑，发现未加锁，
        - 其他客户端 → 执行加锁逻辑
          - 已加锁 → 等待锁释放
          - 未加锁 → 直接加锁
        - 释放锁 → 加锁次数为0时 → 执行lua脚本删除对应的缓存key → LockName = id + threadId , 其他客户端执行unlock时先判断当前threadId是否在字典里
      - 锁类型 → hash key → myLock:"874398-843k0-0835-9884-87774:1" → 里面会保存客户端id，其他客户端不可重入锁 or 删除锁
      - 图解



  - Zookeeper分布式锁
    - Curator框架
      - 基于zookeeper进行优化，实现分布式锁等功能
        - 优化了zookeeper存在的羊群效应 → 即几十个客户端同时竞争一把锁的情况下
          - 锁释放时，需要通知所有的几十个客户端
          - 竞争加锁时，几十个客户端乱哄哄的争抢加锁
        - 保证了在锁释放时，只需要通知顺序子节点即可 → 同时只有顺序节点拿到锁，没有其他几十个客户端争抢锁这种情况出现
      - 原理
        - 加锁
          - 客户端 → 执行加锁逻辑，生成节点 /usr/lock/ → 它是第一个进行加锁的客户端 → 生成临时顺序节点 /usr/lock/******01
          - 其他客户端 → 节点 /usr/lock/已经生成 → 且已经有一个子节点存在 → 生成临时顺序节点 /usr/lock/******02
          - 其他客户端 → 节点 /usr/lock/已经生成 → 且已经有两个子节点存在 → 生成临时顺序节点 /usr/lock/******03
        - 释放锁 → 删除临时顺序节点 → 通知下一个临时顺序节点节点进行加锁
      - 在zookeeper脑裂时仍存在两个客户端同时上锁的可能
      - 图解




  - 分布式锁应对高并发场景
    - 分段加锁+合并扣减 → 相当麻烦的一套逻辑实现
    - 大公司一般都有分布式kv存储 → tair，redis，mongodb，高并发，每秒几万几十万都没问题，甚至每秒百万
      - 实时库存数据放kv存储里去，先查库存再扣减库存，你在操作库存的时候，直接扣减
      - 如果你发现扣减之后是负数的话，此时就认为库存超卖了，回滚刚才的扣减，返回提示给用户
      - 对kv做的库存修改写MQ，异步同步落数据库，相当于异步双写，用分布式kv抗高并发，做好一致性方案